Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System using Modern C++

Dr. Tsung-Wei (TW) Huang

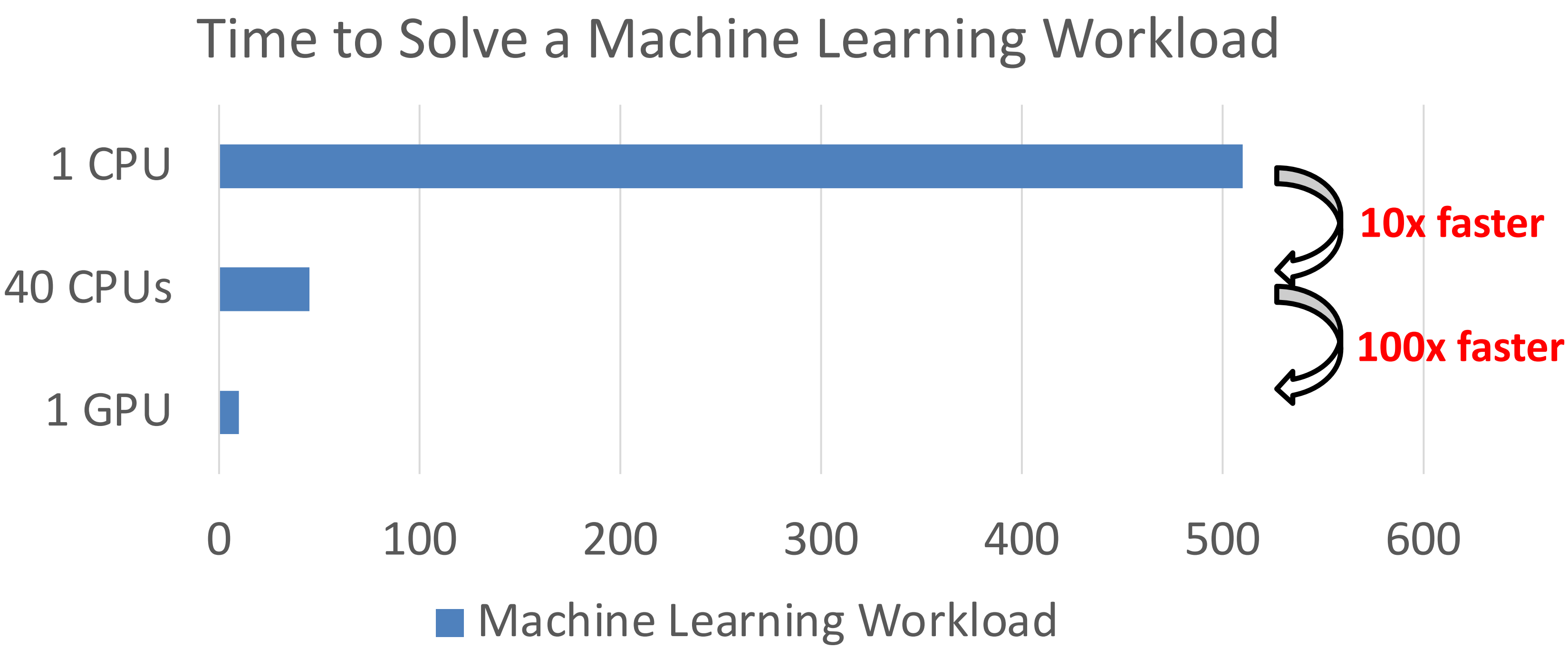
Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

https://taskflow.github.io/

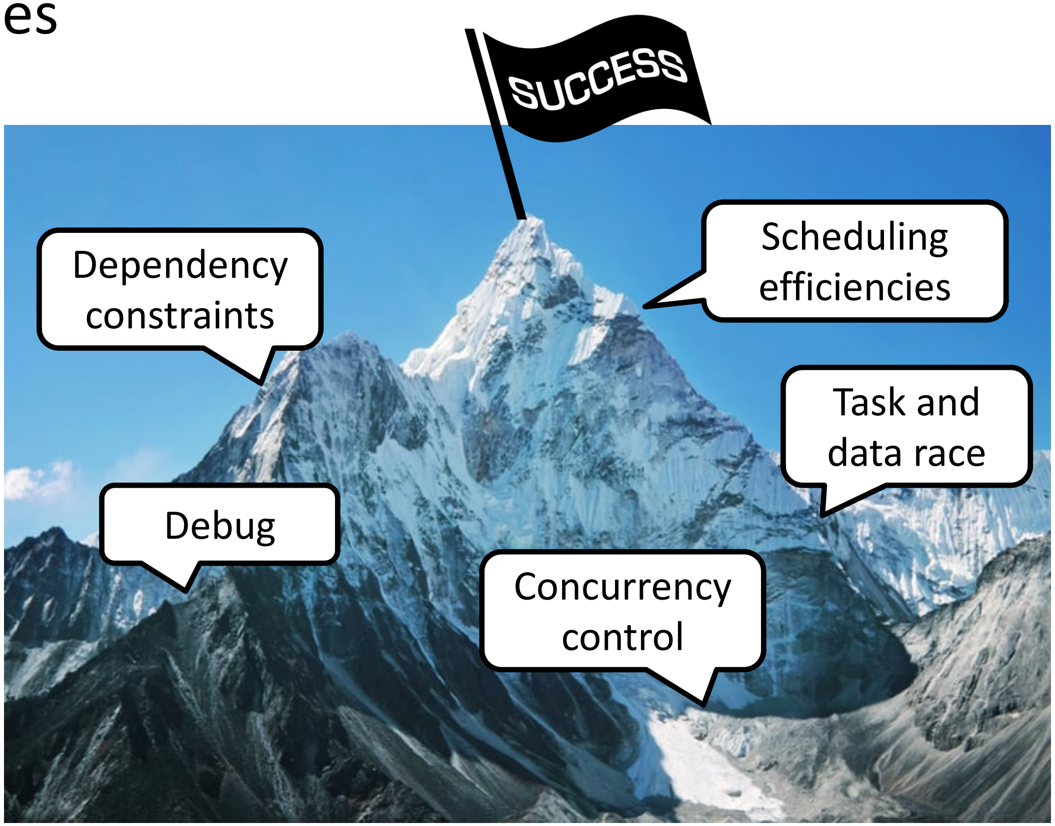
Why Parallel Computing?

- It's critical to advance your application performance



Parallel Programming is Not Easy, Yet

- **You need to deal with many difficult technical details**
  - Standard concurrency control
  - Task dependencies
  - Scheduling
  - Data race
  - … (more)



Many developers have hard time in getting them right!

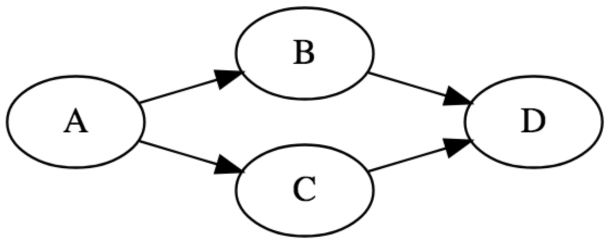
Taskflow offers a solution

How can we make it easier for C++ developers to quickly write parallel and heterogeneous programs with high performance scalability and simultaneous high productivity?

“Hello World” in Taskflow

```
#include <taskflow/taskflow.hpp>   // Taskflow is header-only
int main(){
  tf::Taskflow taskflow;
  tf::Executor executor;
  auto [A, B, C, D] = taskflow.emplace(
    [] () { std::cout << "TaskA\n"; }
    [] () { std::cout << "TaskB\n"; },
    [] () { std::cout << "TaskC\n"; },
    [] () { std::cout << "TaskD\n"; }
  );
  A.precede(B, C);  // A runs before B and C
  D.succeed(B, C);  // D runs after    B and C
  executor.run(taskflow).wait(); // submit the taskflow to the executor
  return 0;
}
```

*Only **15 lines** of code to get a parallel task execution!*



Agenda

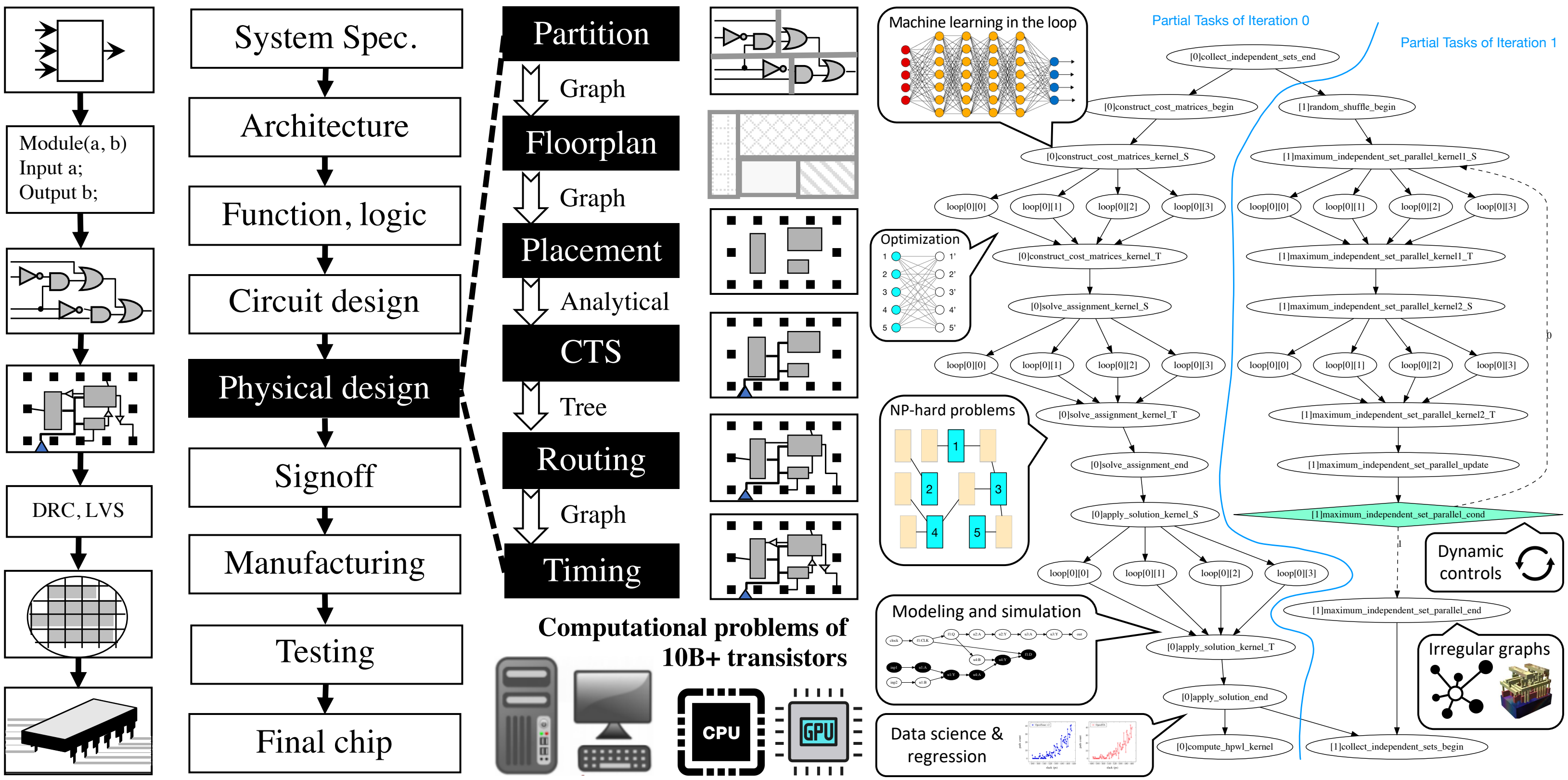
- ❑ **Express your parallelism in the right way**
- ❑ **Parallelize your applications using Taskflow**
- ❑ **Understand our scheduling algorithm**
- ❑ **Boost performance in real applications**
- ❑ **Make C++ amenable to heterogeneous parallelism**

Agenda

- ❑ Express your parallelism in the right way
- ❑ Parallelize your applications using Taskflow
- ❑ Understand our scheduling algorithm
- ❑ Boost performance in real applications
- ❑ Make C++ amenable to heterogeneous parallelism

Motivation: Parallelizing VLSI CAD Tools

- ❑ **Billions of tasks with diverse computational patterns**



How can we write efficient C++ parallel programs for this *monster computational task graph* with ***millions of CPU-GPU dependent tasks along with algorithmic control flow***”

We Invested a lot in Existing Tools …

pthreads
OpenMP
C++
TBB
NVIDIA CUDA
SYCL
PaRSEC
Open MPI
STE||AR GROUP
StarPU
ECP
EXASCALE COMPUTING PROJECT
OpenCL
RAJA
kokkos

Two Big Problems of Existing Tools

- **Our problems define *complex task dependencies***
  - **Example**: analysis algorithms compute the circuit network of million of node and dependencies
  - **Problem**: existing tools are often good at loop parallelism but weak in expressing heterogeneous task graphs at this large scale
- **Our problems define *complex control flow***
  - **Example**: optimization algorithms make essential use of *dynamic control flow* to implement various patterns
    - Combinatorial optimization, analytical methods
  - **Problem**: existing tools are *directed acyclic graph* (DAG)-based and do not anticipate cycles or conditional dependencies, lacking *end-to-end* parallelism

Example: An Iterative Optimizer

- **4 computational tasks with dynamic control flow**

  #1: starts with `init` task

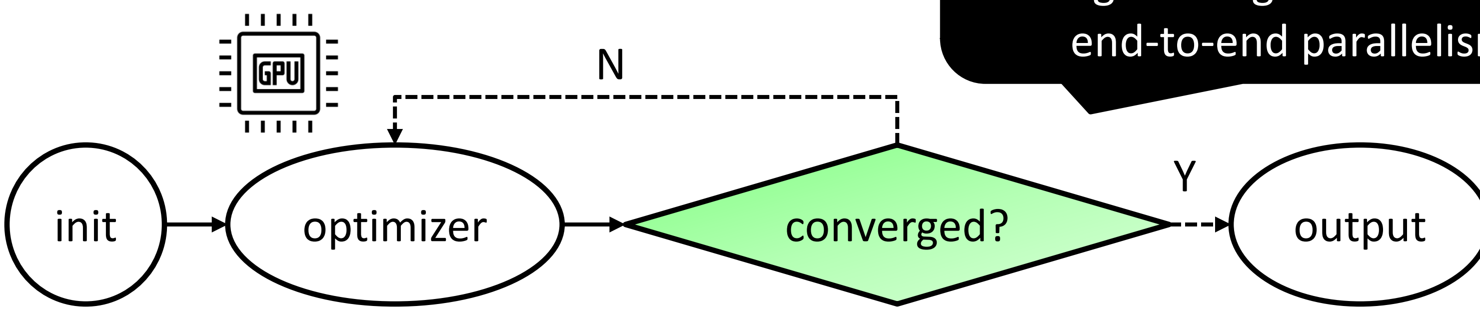
  #2: enters the `optimizer` task (e.g., GPU math solver)

  #3: checks if the optimization converged

  - No: loops back to `optimizer`
  - Yes: proceeds to `stop`

  #4: outputs the result

How can we easily describe this workload of *dynamic control flow* using existing tools to achieve end-to-end parallelism?



Millions of such tasks?

End-to-end parallelism?

Need a New C++ Parallel Programming System

While designing parallel algorithms is non-trivial …

what makes parallel programming an enormous challenge is the infrastructure work of "***how to efficiently express dependent tasks along with an algorithmic control flow and schedule them across heterogeneous computing resources***"

Agenda

- ❑ Express your parallelism in the right way
- ❑ Parallelize your applications using Taskflow
- ❑ Understand our scheduling algorithm
- ❑ Boost performance in real applications
- ❑ Make C++ amenable to heterogeneous parallelism

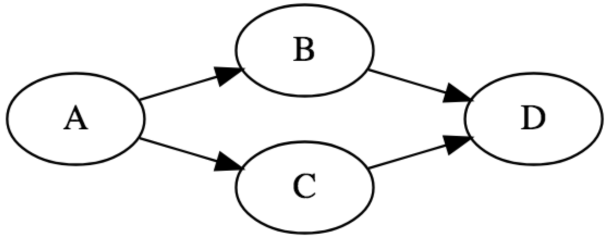
Many arguments are based on my personal opinions – no offense, no criticism, just plain C++ from an end user's perspective

“Hello World” in Taskflow (Revisited)

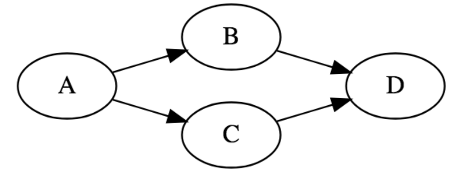
```
#include <taskflow/taskflow.hpp>   // Taskflow is header-only
int main(){
  tf::Taskflow taskflow;
  tf::Executor executor;
  auto [A, B, C, D] = taskflow.emplace(
    [] () { std::cout << "TaskA\n"; }
    [] () { std::cout << "TaskB\n"; },
    [] () { std::cout << "TaskC\n"; },
    [] () { std::cout << "TaskD\n"; }
  );
  A.precede(B, C);  // A runs before B and C
  D.succeed(B, C);  // D runs after    B and C
  executor.run(taskflow).wait(); // submit the taskflow to the executor
  return 0;
}
```

Taskflow defines five tasks:

1. static task
2. dynamic task
3. cudaFlow task
4. condition task
5. module task



“Hello World” in OpenMP




```
#include <omp.h>  // OpenMP is a lang ext to describe parallelism using compiler directives
int main(){
  #omp parallel num_threads(std::thread::hardware_concurrency())
  {
    int A_B, A_C, B_D, C_D;
    #pragma omp task depend(out: A_B, A_C)
    {
      s t d : : c o u t << ”TaskA\n” ;
    }
    #pragma omp task depend(in: A_B; out: B_D)
    {
      s t d : : c o u t << ” TaskB\n” ;
    }
    #pragma omp task depend(in: A_C; out: C_D)
    {
      s t d : : c o u t << ” TaskC\n” ;
    }
    #pragma omp task depend(in: B_D, C_D)
    {
      s t d : : c o u t << ”TaskD\n” ;
    }
  }
  return 0;
}
```

Task dependency clauses

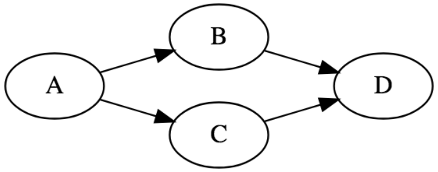
Task dependency clauses

Task dependency clauses

Task dependency clauses

OpenMP task clauses are static and explicit;
Programmers are responsible for a proper order of writing tasks consistent with sequential execution

“Hello World” in TBB




```
#include <tbb.h>  // Intel’s TBB is a general-purpose parallel programming library in C++
int main(){
   using namespace tbb;
   using namespace tbb:flow;
   int n = task_scheduler init::default_num_threads () ;
   task scheduler_init init(n);
   graph g;
   continue_node<continue_msg> A(g, [] (const continue msg &) {
      s t d : : c o u t << “TaskA” ;
   }) ;
   continue_node<continue_msg> B(g, [] (const continue msg &) {
      s t d : : c o u t << “TaskB” ;
   }) ;
   continue_node<continue_msg> C(g, [] (const continue msg &) {
      s t d : : c o u t << “TaskC” ;
   }) ;
   continue_node<continue_msg> C(g, [] (const continue msg &) {
      s t d : : c o u t << “TaskD” ;
   }) ;
   make_edge(A, B);
   make_edge(A, C);
   make_edge(B, D);
   make_edge(C, D);
   A.try_put(continue_msg());
   g.wait_for_all();
}
```

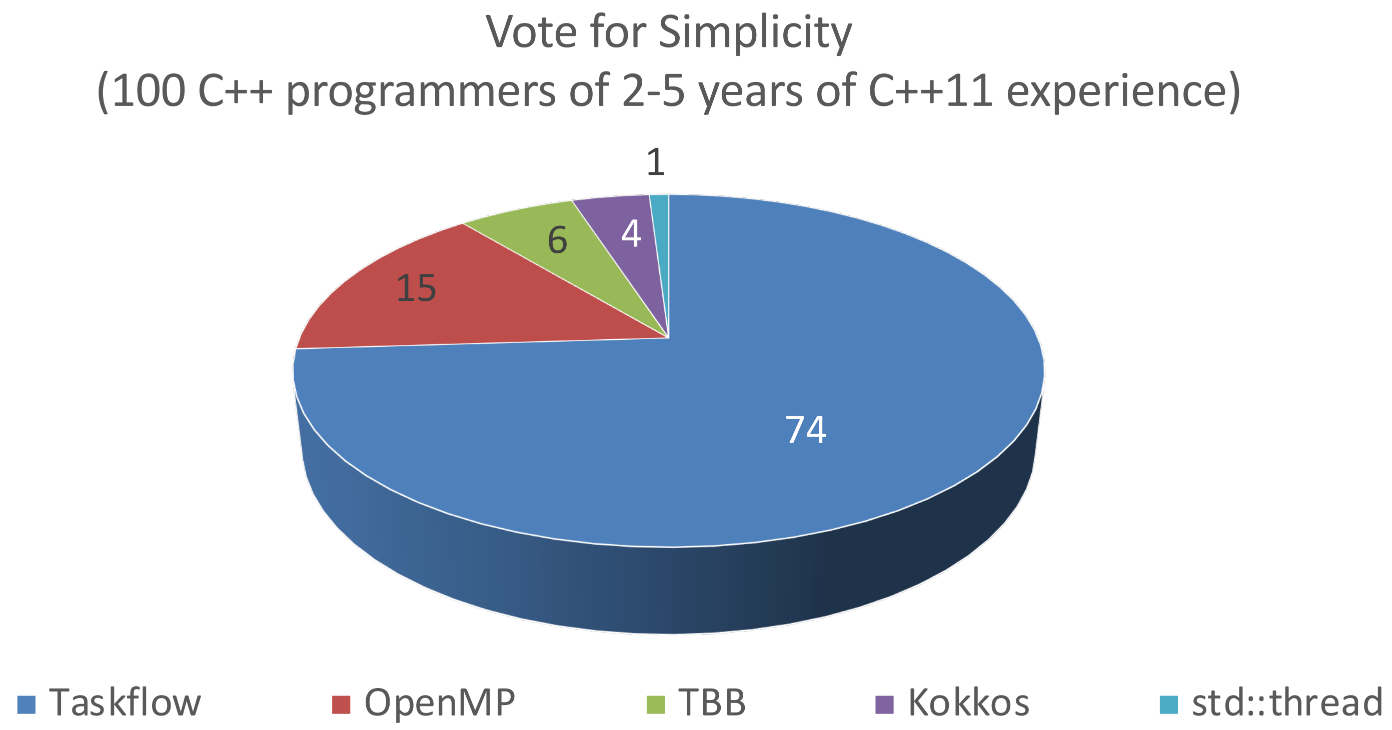
Use TBB’s FlowGraph for task parallelism

Declare a task as a continue_node

TBB has excellent performance in generic parallel computing. Its drawback is mostly in the ease-of-use standpoint (simplicity, expressivity, and programmability).

TBB FlowGraph: https://software.intel.com/content/www/us/en/develop/home.html

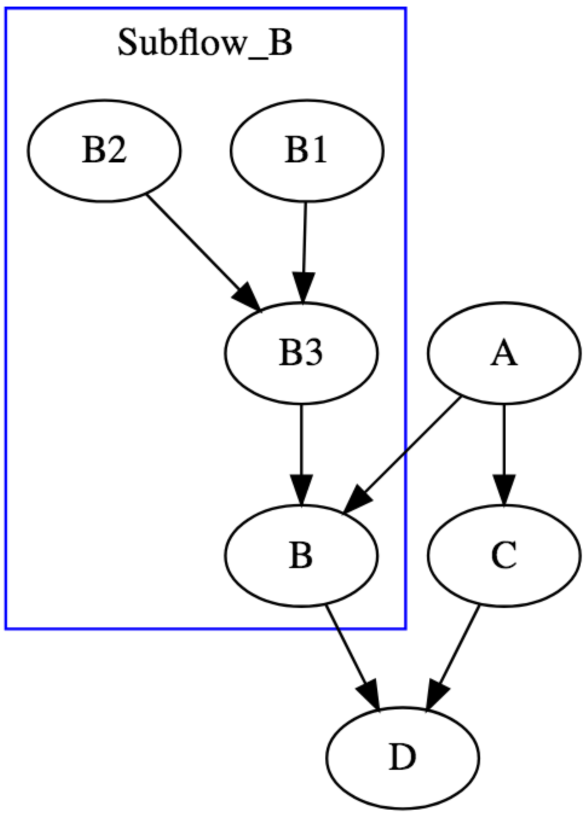
"Hello World" Summary (Less Biased)



#1 concern: *"My application is already very complex; it's important the parallel programming library doesn't become another burden."*

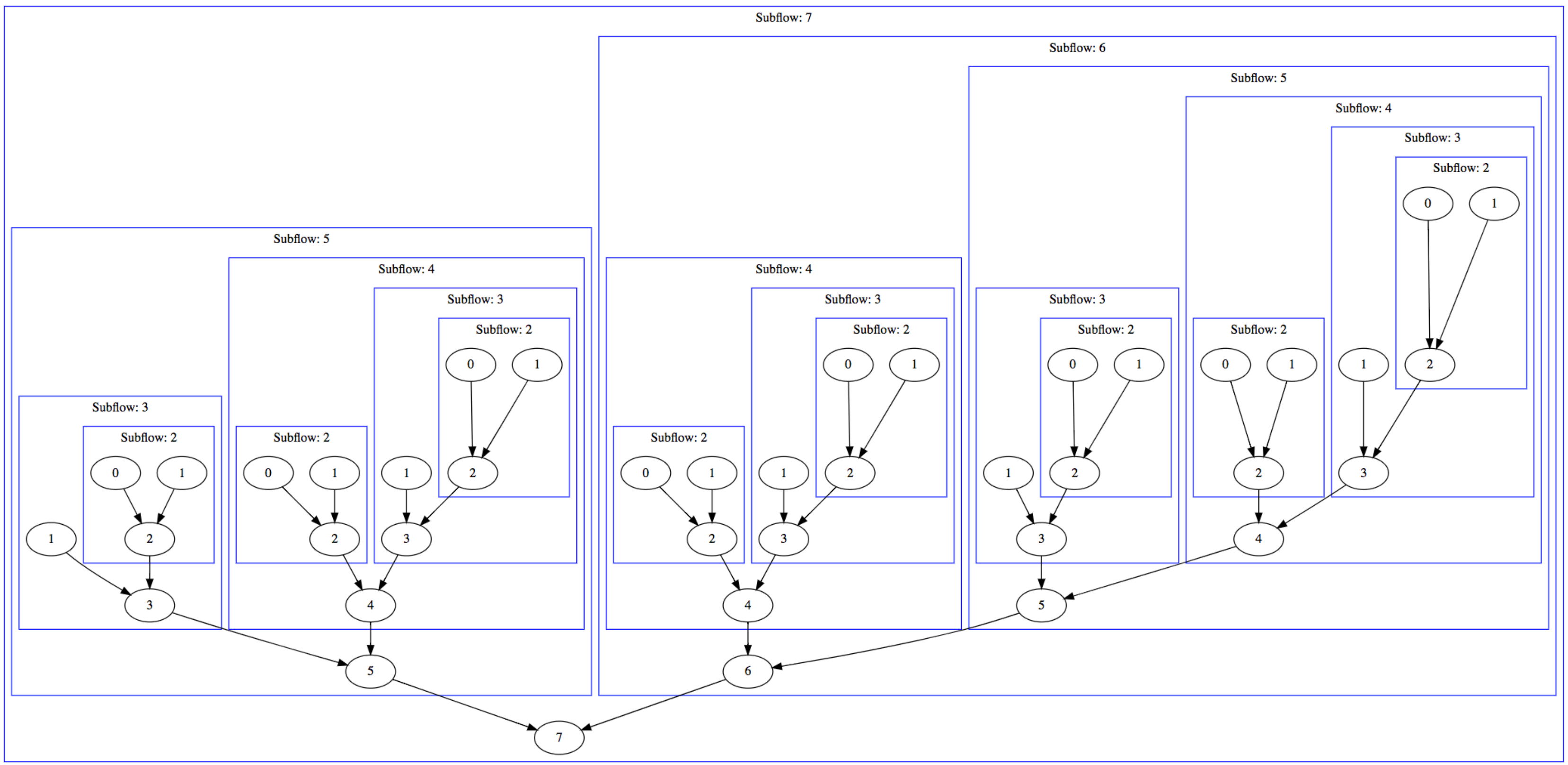
#2: Dynamic Tasking (Subflow)

```
// create three regular tasks
tf::Task A = tf.emplace([](){}).name("A");
tf::Task C = tf.emplace([](){}).name("C");
tf::Task D = tf.emplace([](){}).name("D");

// create a subflow graph (dynamic tasking)
tf::Task B = tf.emplace([] (tf::Subflow& subflow) {
  tf::Task B1 = subflow.emplace([](){}).name("B1");
  tf::Task B2 = subflow.emplace([](){}).name("B2");
  tf::Task B3 = subflow.emplace([](){}).name("B3");
  B1.precede(B3);
  B2.precede(B3);
}).name("B");

A.precede(B); // B runs after A
A.precede(C); // C runs after A
B.precede(D); // D runs after B
C.precede(D); // D runs after C
```



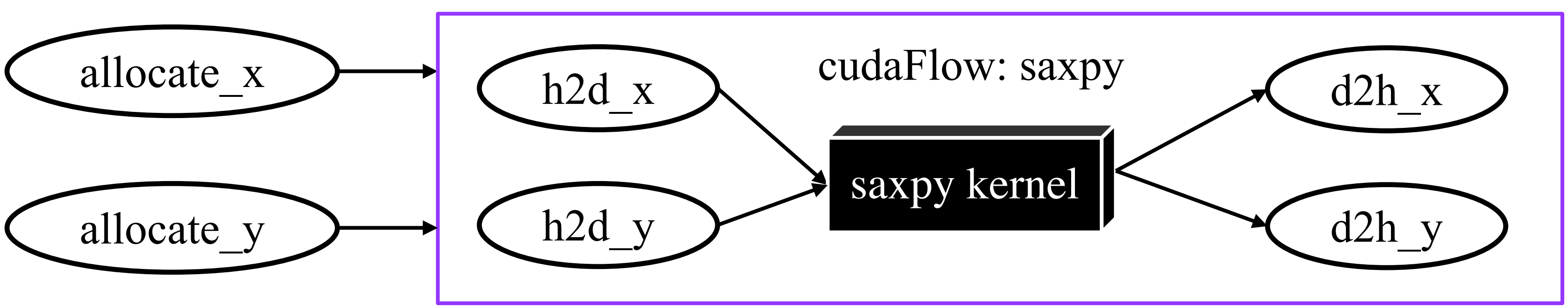
Subflow can be Nested

- **Find the 7th Fibonacci number using subflow**
  - Fib(n) = Fib(n-1) + Fib(n-2)



#3: Heterogeneous Tasking (cudaFlow)

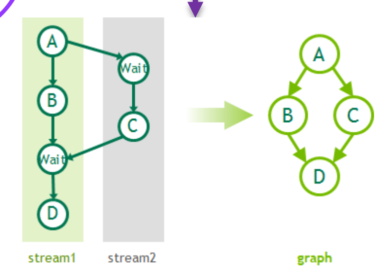
- Single Precision AX + Y (“SAXPY”)
  - Get x and y vectors on CPU (allocate_x, allocate_y)
  - Copy x and y to GPU (h2d_x, h2d_y)
  - Run saxpy kernel on x and y (saxpy kernel)
  - Copy x and y back to CPU (d2h_x, d2h_y)



Heterogeneous Tasking (cont'd)

```
const unsigned N = 1<<20;
std::vector<float> hx(N, 1.0f), hy(N, 2.0f);
float *dx{nullptr}, *dy{nullptr};
auto allocate_x = taskflow.emplace([&](){ cudaMalloc(&dx, 4*N);});
auto allocate_y = taskflow.emplace([&](){ cudaMalloc(&dy, 4*N);});

auto cudaflow = taskflow.emplace([&](tf::cudaFlow& cf) {
   auto h2d_x = cf.copy(dx, hx.data(), N);  // CPU-GPU data transfer
   auto h2d_y = cf.copy(dy, hy.data(), N);
   auto d2h_x = cf.copy(hx.data(), dx, N);  // GPU-CPU data transfer
   auto d2h_y = cf.copy(hy.data(), dy, N);
   auto kernel = cf.kernel((N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy);
   kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
});

cudaflow.succeed(allocate_x, allocate_y);
executor.run(taskflow).wait();
```

To Nvidia ***cudaGraph***



Three Key Motivations

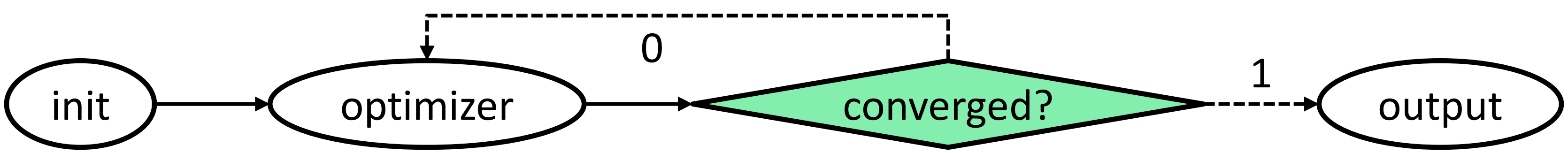
- **Our closure enables stateful interface**
  - Users capture data in reference to marshal data exchange between CPU and GPU tasks
- **Our closure hides implementation details judiciously**
  - We use cudaGraph (since cuda 10) due to its excellent performance, much faster than streams in large graphs
- **Our closure extend to new accelerator types**
  - syclFlow, openclFlow, coralFlow, tpuFlow, fpgaFlow, etc.

```
auto cudaflow = taskflow.emplace([&](tf::cudaFlow& cf) {
  auto h2d_x = cf.copy(dx, hx.data(), N);  // CPU-GPU data transfer
  auto h2d_y = cf.copy(dy, hy.data(), N);
  auto d2h_x = cf.copy(hx.data(), dx, N);  // GPU-CPU data transfer
  auto d2h_y = cf.copy(hy.data(), dy, N);
  auto kernel = cf.kernel((N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy);
  kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
});
```

We do not simplify kernel programming but **focus on *CPU-GPU tasking* that affects the performance to a large extent!** (same for data abstraction)

#4: Conditional Tasking

```
auto init          = taskflow.emplace([&](){ initialize_data_structure(); } )
                               .name(”init");
auto optimizer     = taskflow.emplace([&](){ matrix_solver(); } )
                               .name(”optimizer");
auto converged     = taskflow.emplace([&](){ return converged() ? 1 : 0 } )
                               .name(”converged");
auto output        = taskflow.emplace([&](){ std::cout << ”done!\n"; } );
                               .name(”output");
init.precede(optimizer);
optimizer.precede(converged);
converged.precede(optimizer, output);  // return 0 to the optimizer again
```



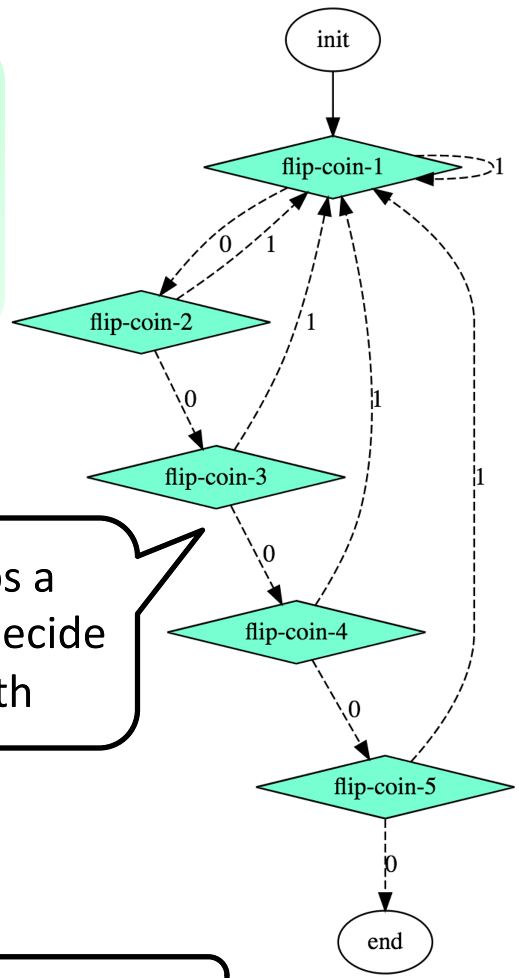
Tip!

Condition task integrates control flow into a task graph to form end-to-end parallelism; in this example, there are ultimately four tasks ever created

Conditional Tasking (cont'd)

```
auto A = taskflow.emplace([&](){ } );
auto B = taskflow.emplace([&](){ return rand()%2; } );
auto C = taskflow.emplace([&](){ return rand()%2; } );
auto D = taskflow.emplace([&](){ return rand()%2; } );
auto E = taskflow.emplace([&](){ return rand()%2; } );
auto F = taskflow.emplace([&](){ return rand()%2; } );
auto G = taskflow.emplace([&](){});

A.precede(B).name("init");
B.precede(C, B).name("flip-coin-1");
C.precede(D, B).name("flip-coin-2");
D.precede(E, B).name("flip-coin-3");
E.precede(F, B).name("flip-coin-4");
F.precede(G, B).name("flip-coin-5");
G.name("end");
```

Each task flips a binary coin to decide the next path



Tip!

You can describe non-deterministic, nested control flow!

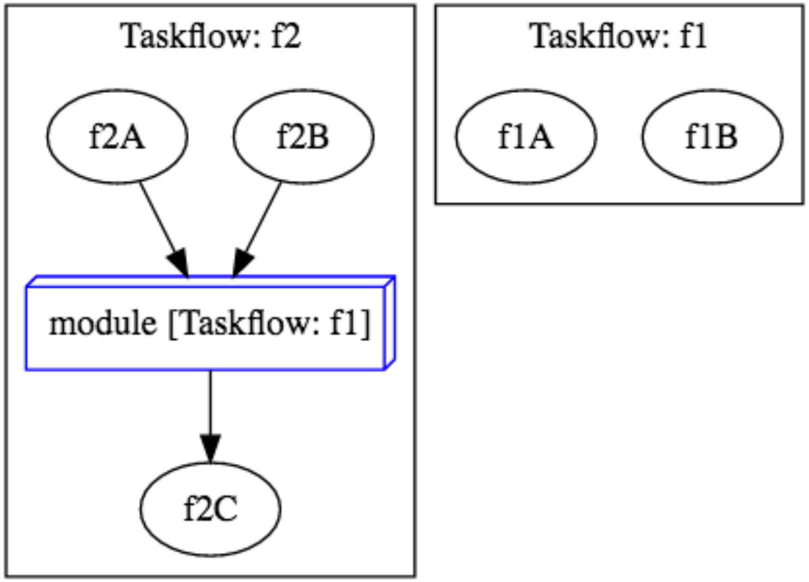
Existing Frameworks on Control Flow?

- **Expand a task graph across fixed-length iterations**
  - Graph size is linearly proportional to decision points
- **Unknown iterations? Non-deterministic conditions?**
  - Complex dynamic tasks executing “if” on the fly
- **Dynamic control flows and dynamic tasks?**
- **… (resort to client-side decision)**

Existing frameworks on expressing conditional tasking or dynamic control flow suffer from exponential growth of code complexity

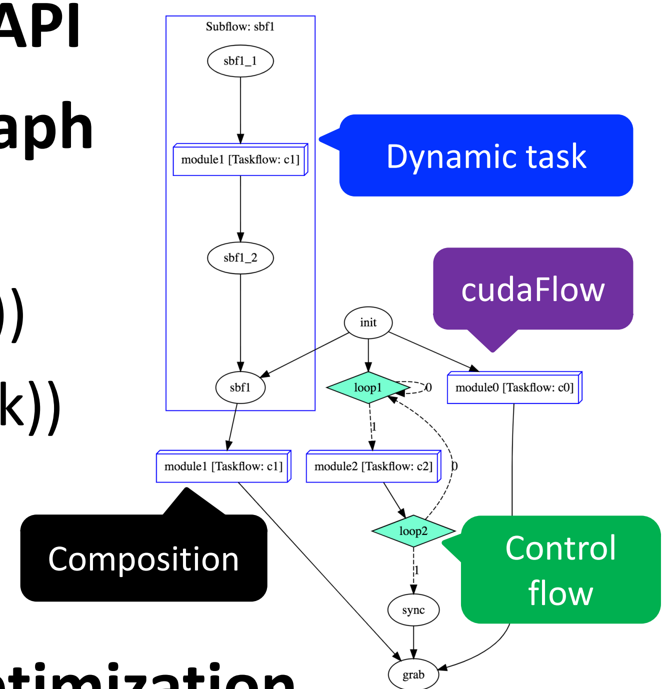
#5: Composable Tasking

```
tf::Taskflow f1, f2;

auto [f1A, f1B] = f1.emplace(
  [](){ std::cout << "Task f1A\n"; },
  [](){ std::cout << "Task f1B\n"; }
);
auto [f2A, f2B, f2C] = f2.emplace(
  [](){ std::cout << "Task f2A\n"; },
  [](){ std::cout << "Task f2B\n"; },
  [](){ std::cout << "Task f2C\n"; }
);

auto f1_module_task = f2.composed_of(f1);

f1_module_task.succeed(f2A, f2B)
              .precede(f2C);
```

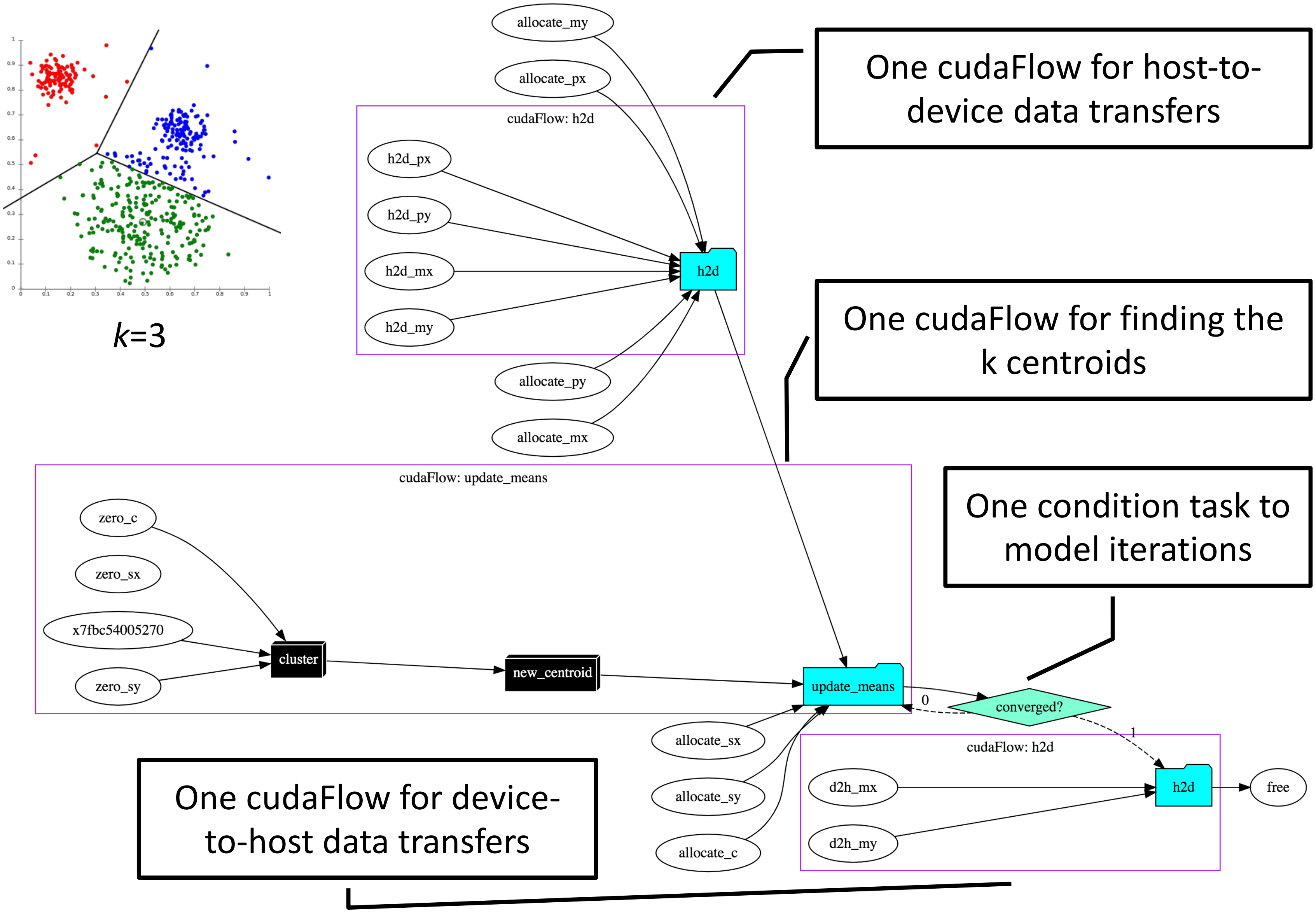


Everything is Unified in Taskflow

- **Use “`emplace`” to create a task**
- **Use “`precede`” to add a task dependency**
- **No need to learn different sets of API**
- **You can create a really complex graph**
  - Subflow(ConditionTask(cudaFlow))
  - ConditionTask(StaticTask(cudaFlow))
  - Composition(Subflow(ConditionTask))
  - Subflow(ConditionTask(cudaFlow))
  - …
- **Scheduler performs end-to-end optimization**
  - Runtime, energy efficiency, and throughput



Example: k-means Clustering



Agenda

- ❑ **Express your parallelism in the right way**
- ❑ **Parallelize your applications using Taskflow**
- ❑ **Understand our scheduling algorithm**
- ❑ **Boost performance in real applications**
- ❑ **Make C++ amenable to heterogeneous parallelism**

Submit Taskflow to Executor

- **Executor manages a set of threads to run taskflows**
  - All execution methods are *non-blocking*
  - All execution methods are *thread-safe*

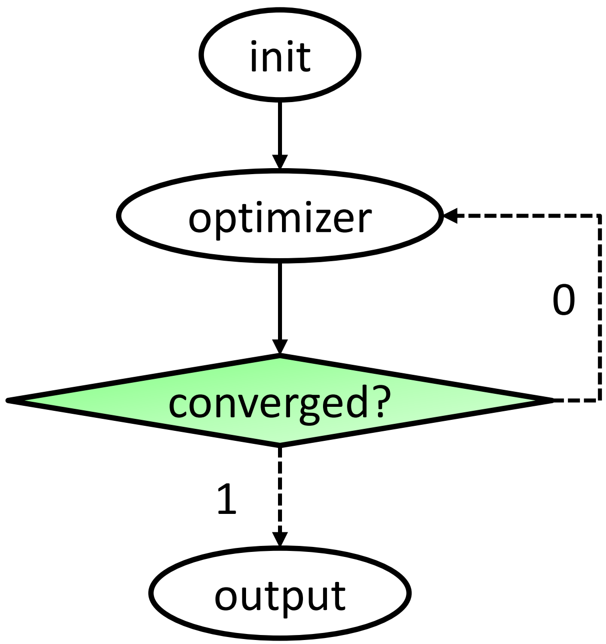
```
{
  tf::Taskflow taskflow1, taskflow2, taskflow3;
  tf::Executor executor;
  // create tasks and dependencies
  // …
  auto  future1 = executor.run(taskflow1);
  auto  future2 = executor.run_n(taskflow2, 1000);
  auto  future3 = executor.run_until(taskflow3, [i=0](){ return i++>5 });
  executor.async([](){ std::cout << “async task\n"; });
  executor.wait_for_all(); // wait for all the above tasks to finish
}
```

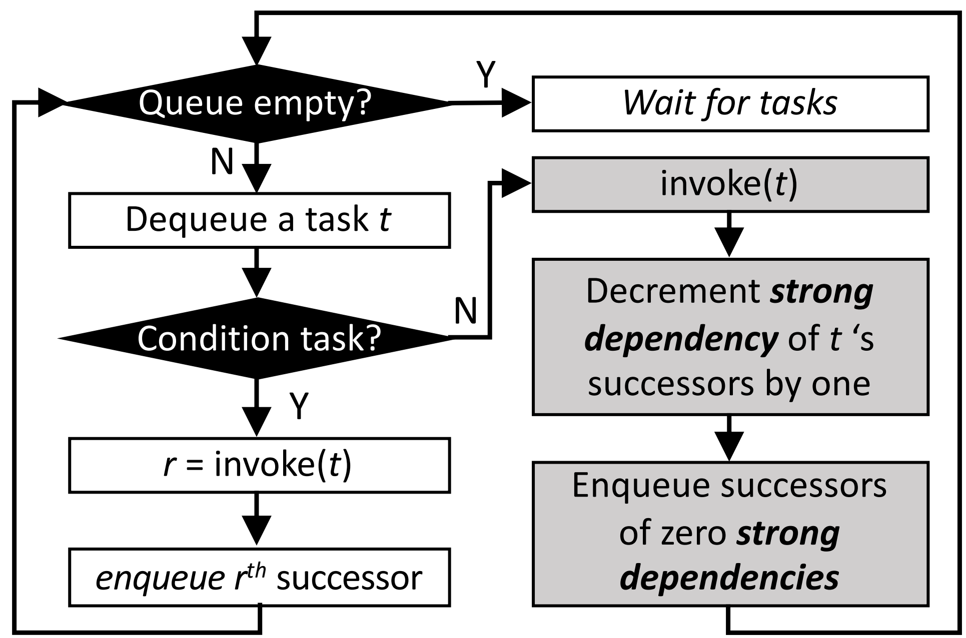
Executor Scheduling Algorithm

- **Task-level scheduling**
  - Decides how tasks are enqueued under control flow
    - Goal #1: ensures a feasible path to carry out control flow
    - Goal #2: avoids task race under cyclic and conditional execution
    - Goal #3: maximizes the capability of conditional tasking
- **Worker-level scheduling**
  - Decides how tasks are executed by which workers
    - Goal #1: adopts work stealing to dynamically balance load
    - Goal #2: adapts workers to available task parallelism
    - Goal #3: maximizes performance, energy, and throughput

Task-level Scheduling

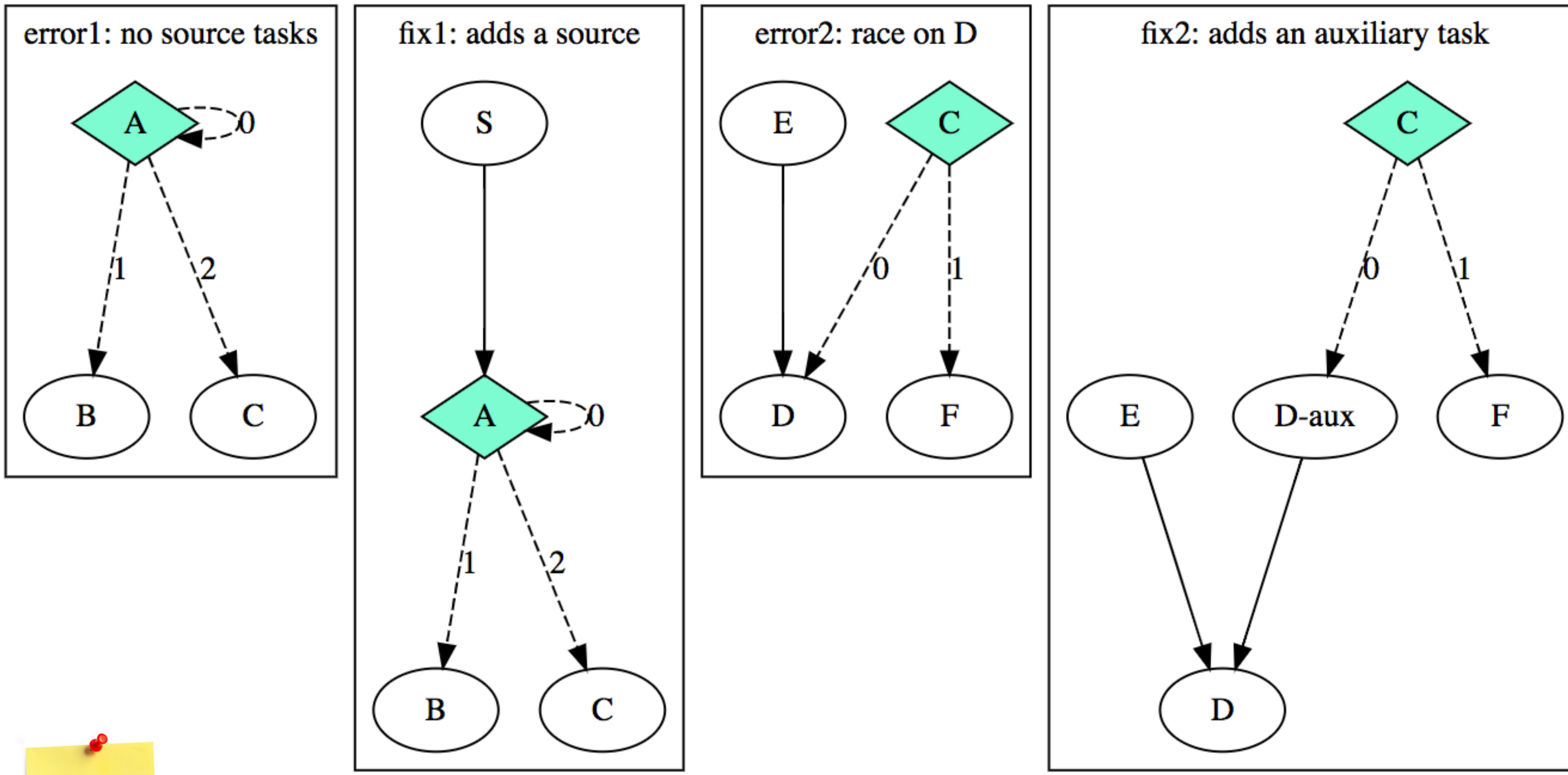
- ***"Strong dependency"* versus *"Weak dependency"***
  - Weak dependency: dependencies out of condition tasks
  - Strong dependency: others else





Task-level Scheduling (cont'd)

- Condition task is powerful but prone to mistakes …

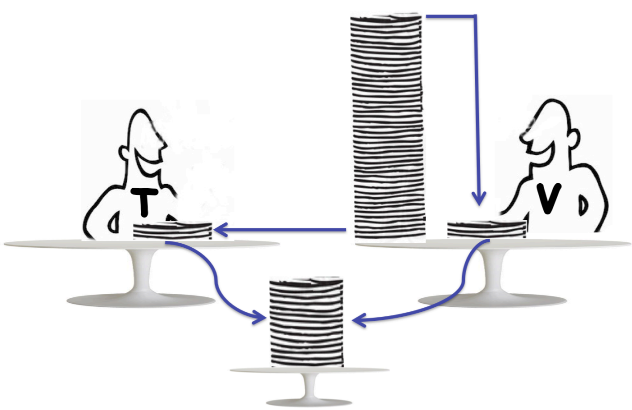


Tip!

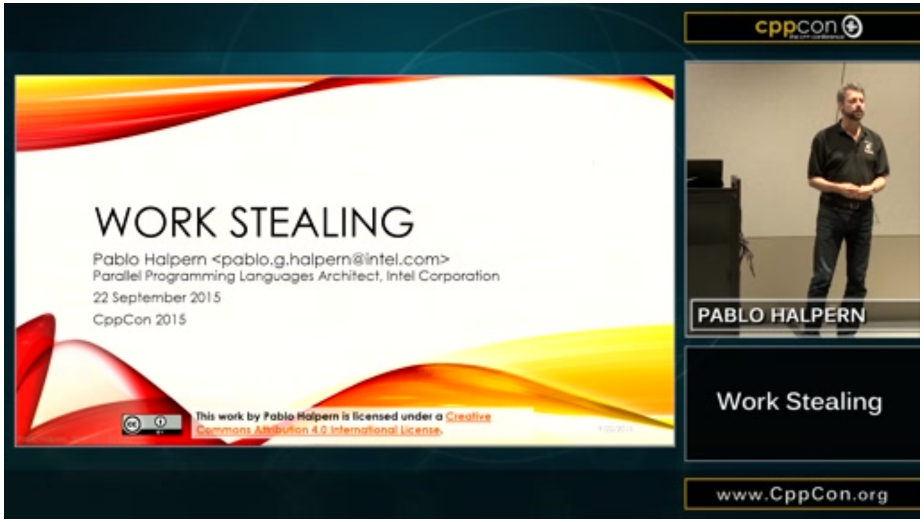
It is users' responsibility to ensure a taskflow is properly conditioned, i.e., no task race under our task-level scheduling policy

Worker-level Scheduling

- **Taskflow adopts *work stealing* to run tasks**
- **What is work stealing?**
  - I finish my jobs first, and then steal jobs from you
  - Improve performance through dynamic load balancing



Work stealing is commonly adopted by parallel task programming libraries (e.g., TBB, StarPU, TPL)



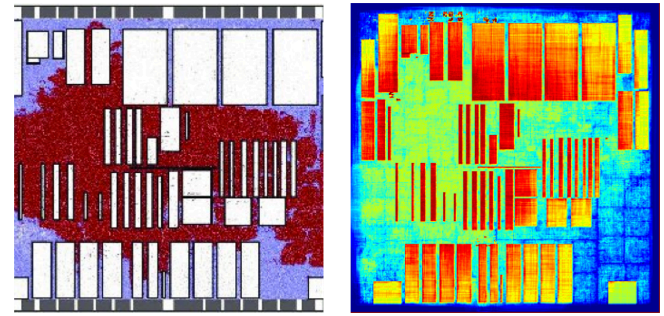
CppCon 2015: Pablo Halpern "Work Stealing," https://www.youtube.com/watch?v=iLHNF7SgVN4

Agenda

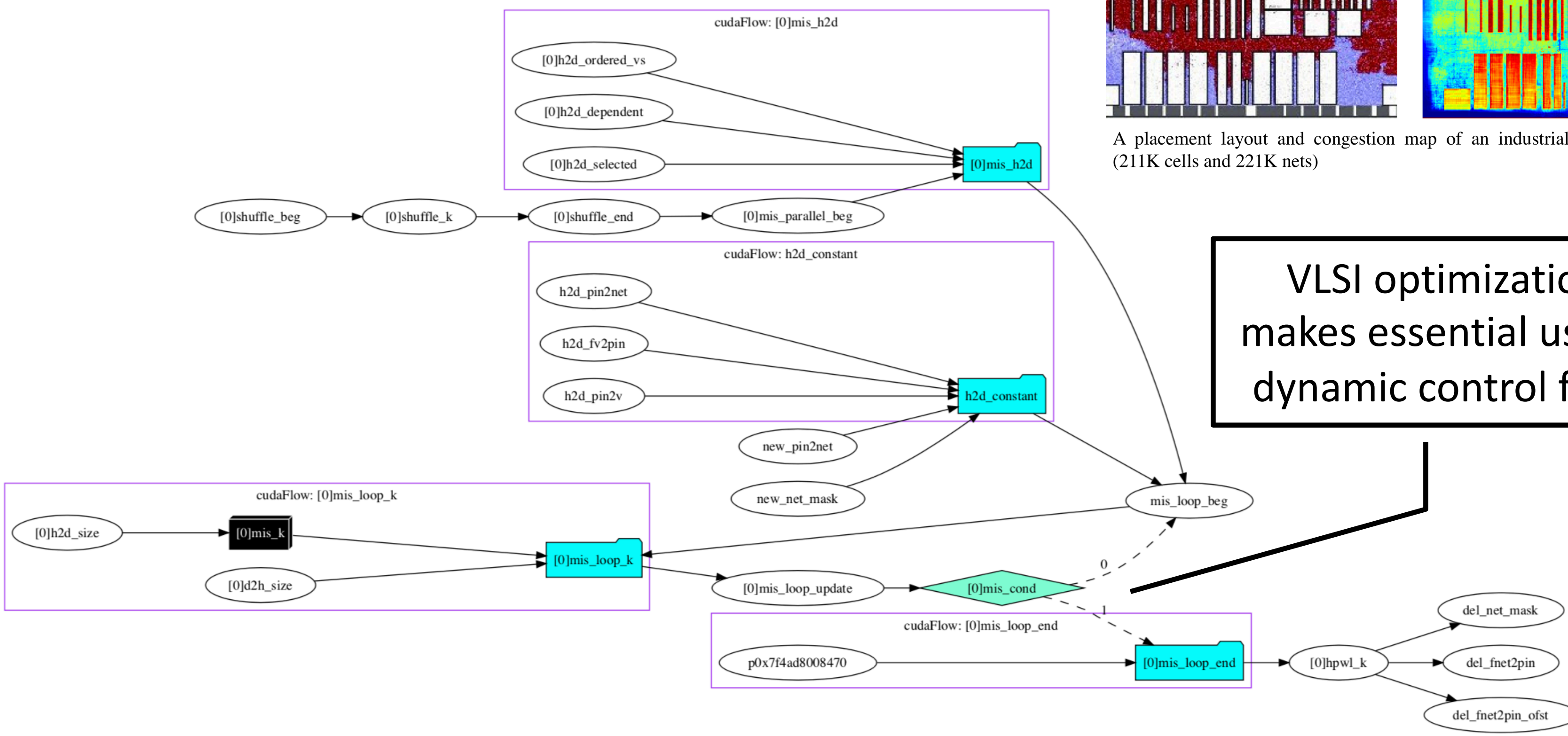
- ❑ Express your parallelism in the right way
- ❑ Parallelize your applications using Taskflow
- ❑ Understand our scheduling algorithm
- ❑ Boost performance in real applications
- ❑ Make C++ amenable to heterogeneous parallelism

Application 1: VLSI Placement

- Optimize cell locations on a chip



A placement layout and congestion map of an industrial circuit, adaptec1 (211K cells and 221K nets)

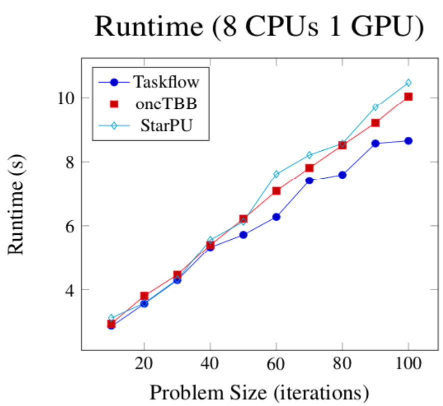


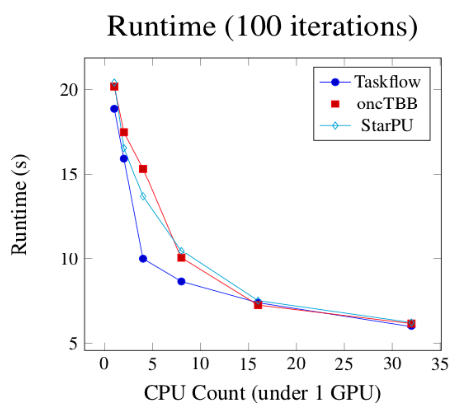
VLSI optimization makes essential use of dynamic control flow

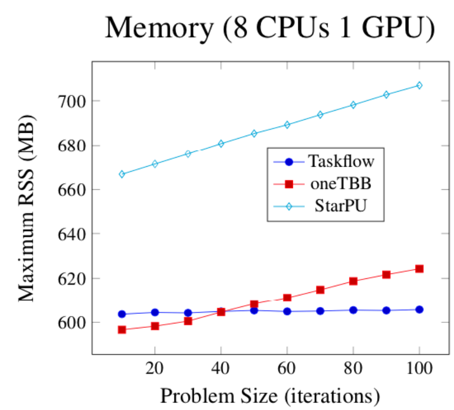
A partial TDG of 4 cudaFlows, 1 conditioned cycle, and 12 static tasks

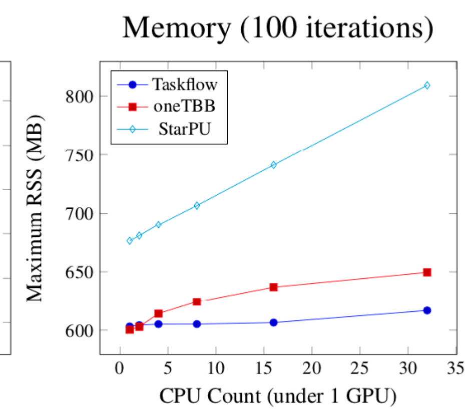
Application 1: VLSI Placement (cont'd)

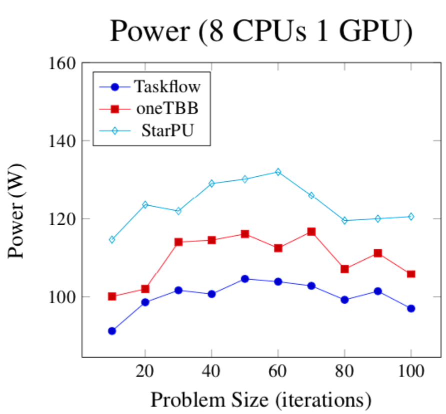
- Runtime, memory, power, and throughput

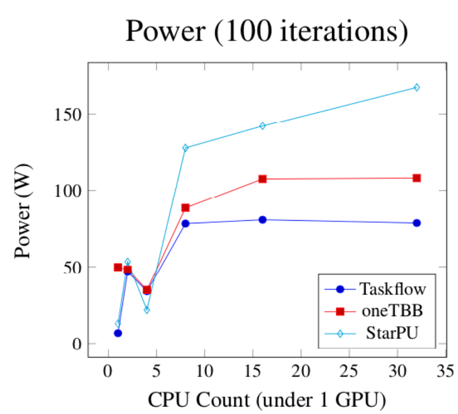








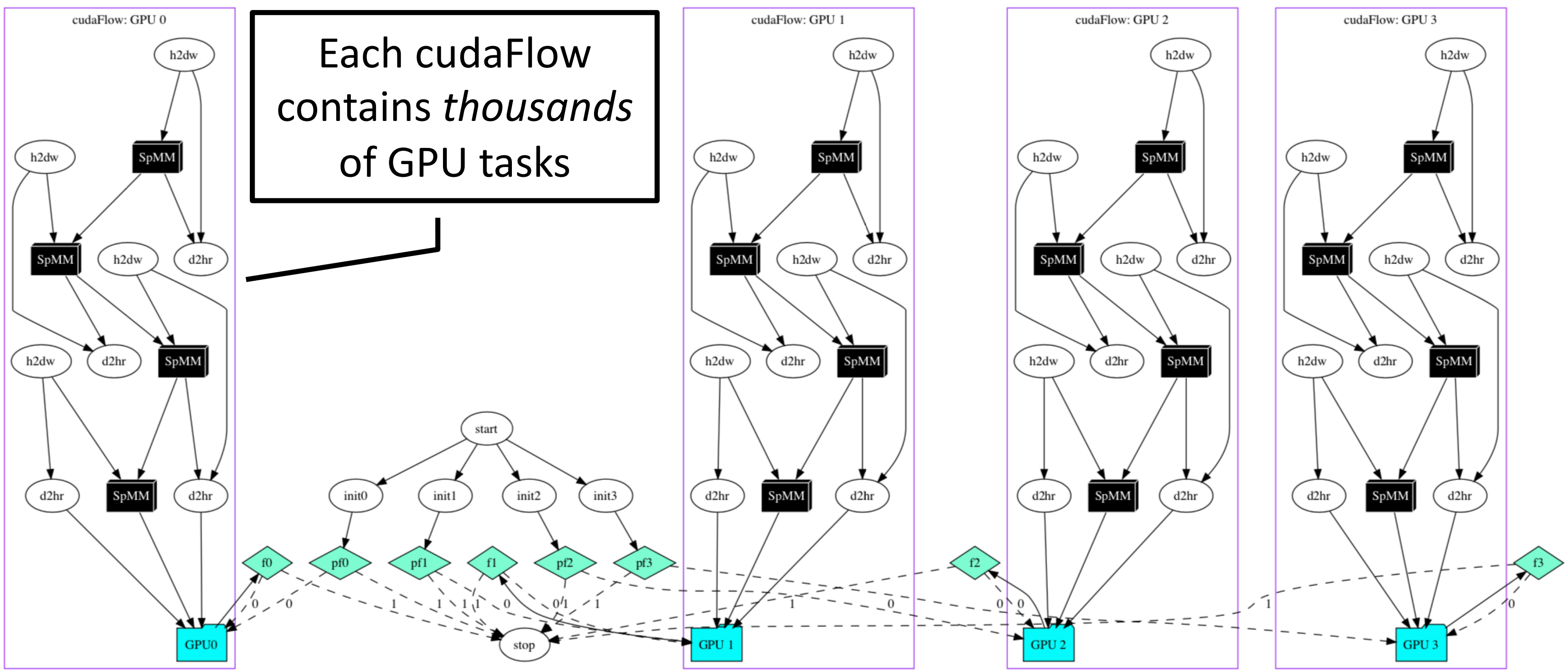




Performance improvement comes from *end-to-end* expression of CPU-GPU dependent tasks using condition tasks

Application 2: Machine Learning

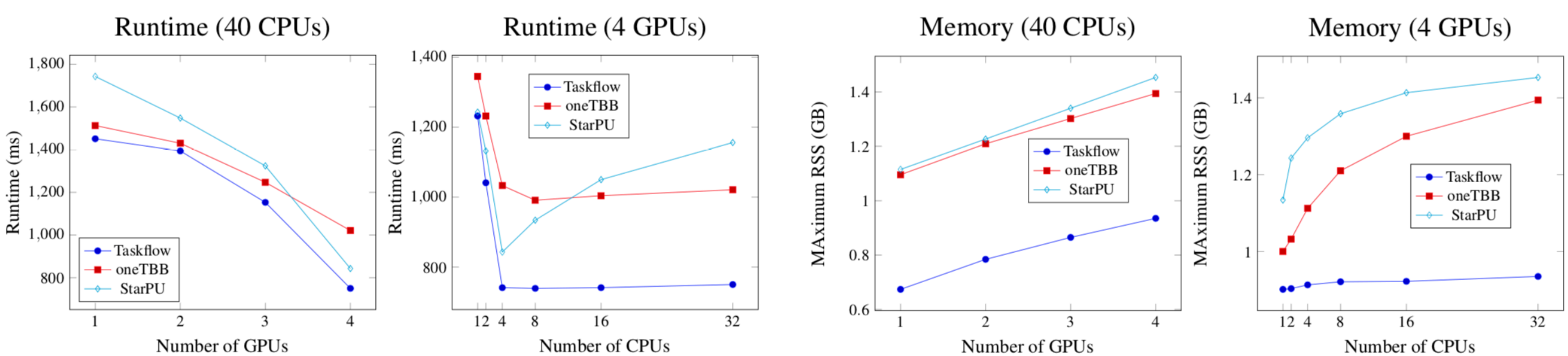
- **Compute a 1920-layer DNN each of 65536 neurons**
  - IEEE HPEC 2020 Neural Network Challenge Compute



A partial taskflow graph of 4 cudaFlows, 6 static tasks, and 8 conditioned cycles for this workload

Application 2: Machine Learning (cont'd)

- **Comparison with TBB and StarPU**
  - Unroll task graphs across iterations found in hindsight
  - Implement cudaGraph for all

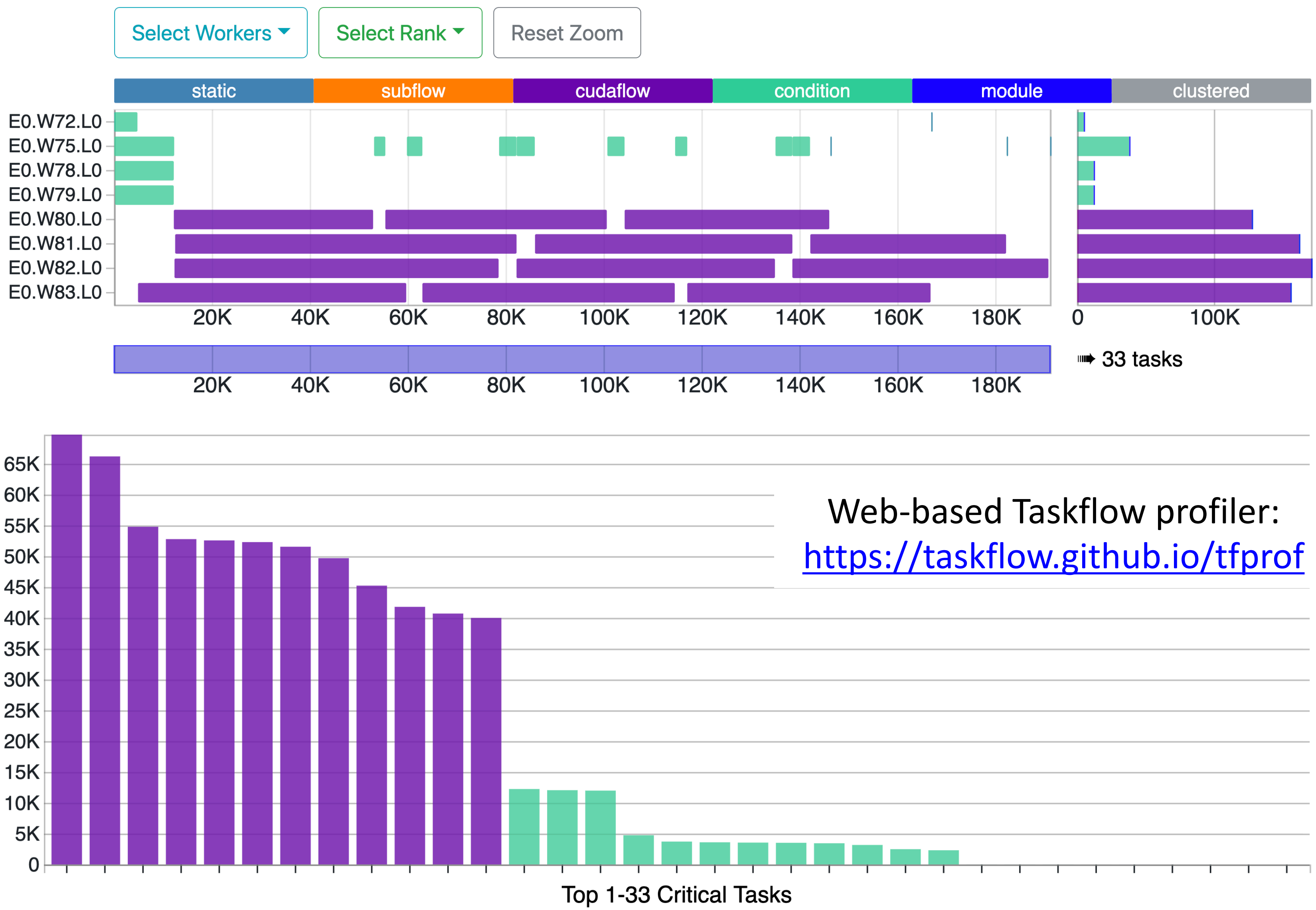


- **Taskflow's runtime is up to 2x faster**
- **Taskflow's memory is up to 1.6x less**

Due to the conditional tasking

Champions of HPEC 2020 Graph Challenge: https://graphchallenge.mit.edu/champions

Application 2: Machine Learning (cont’d)



Web-based Taskflow profiler:
https://taskflow.github.io/tfprof

<u>Parallel programming infrastructure matters</u>

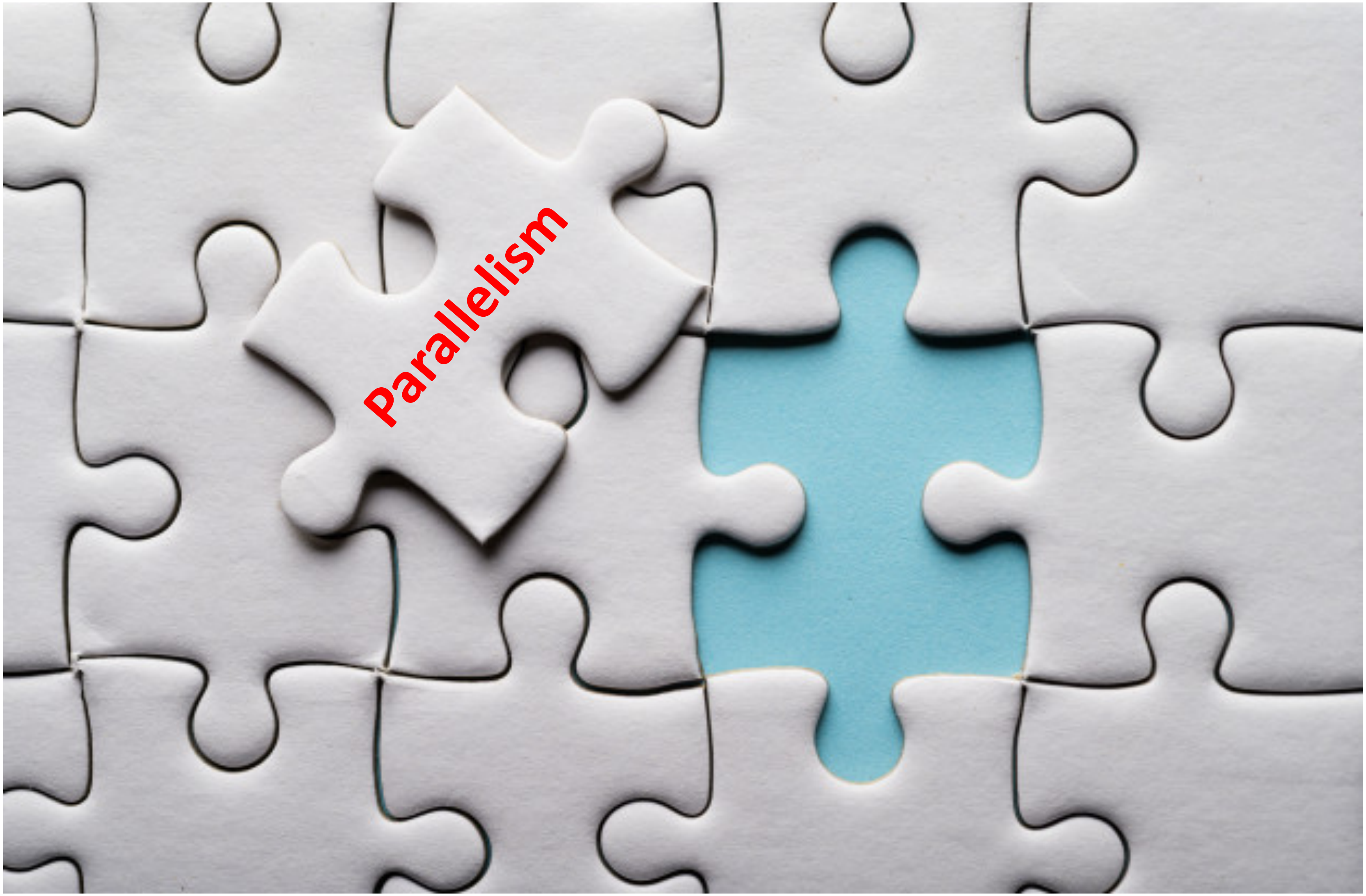
Different models give different implementations. The parallel code/algorithm may run fast, yet the parallel computing infrastructure to support that algorithm may dominate the entire performance.

Taskflow enables *end-to-end* expression of CPU-GPU dependent tasks along with algorithmic control flow

Agenda

- ❑ Express your parallelism in the right way
- ❑ Parallelize your applications using Taskflow
- ❑ Understand our scheduling algorithm
- ❑ Boost performance in real applications
- ❑ Make C++ amenable to heterogeneous parallelism

Parallel Computing is Never Standalone

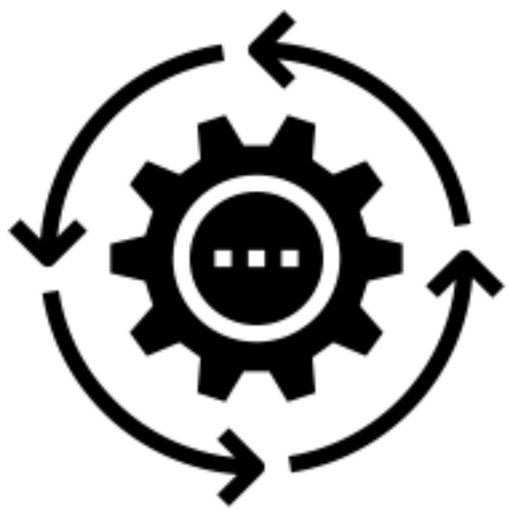


No One Can Express All Parallelisms …

- **Languages** ∪ **Compilers** ∪ **Libraries** ∪ **Programmers**

IMHO, C++ Parallelism Needs Enhancement

- **C++ parallelism is primitive (but in a good shape)**
  - `std::thread` is powerful but very low-level
  - `std::async` leaves off handling task dependencies
  - No easy ways to describe control flow in parallelism
    - C++17 parallel STL count on bulk synchronous parallelism
  - No standard ways to offload tasks to accelerators (GPU)



Conclusion

- **Taskflow is a general-purpose parallel tasking tool**
  - Simple, efficient, and transparent tasking models
  - Efficient heterogeneous work-stealing executor
  - Promising performance in large-scale ML and VLSI CAD
- **Taskflow is not to replace anyone but to**
  - Complement the current state-of-the-art
  - Leverage modern C++ to express task graph parallelism
- **Taskflow is very open to collaboration**
  - We want to integrate OpenCL, SYCL, Intel DPC++, etc.
  - We want to provide higher-level algorithms
  - We want to broaden real use cases

Thank You All Using Taskflow!

Thank You

Dr. Tsung-Wei Huang

tsung-wei.huang@utah.edu

Taskflow: https://taskflow.github.io